REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Dec 94	

4. TITLE AND SUBTITLE
A Demonstration of Client/Server Technology Using Remote Procedure Calls for an Application of File Migration for Moving Records based on Location

5. FUNDING NUMBERS

6. AUTHOR(S)

Andrea Miller

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

AFIT Students Attending:

Arizona State University

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/CI/CIA

94-153

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DEPRTMENT OF THE AIR FORCE
AFIT/CI
2950 P STREET
WRIGHT-PATTERSON AFB OH 45433-7765

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release IAW 190-1
Distribution Unlimited
MICHAEL M. BRICKER, SMSgt, USAF
Chief Administration

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

19950103 035

DTIC QUALITY INSPECTED 3

14. SUBJECT TERMS			15. NUMBER OF PAGES
			37
			16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Title: **A Demonstration of Client/Server Technology**
       Using Remote Procedure Calls
       For An Application of File Migration
       For Moving Records Based on Location

Author: **Andrea Miller**
Rank   : 2nd Lieutenant, USAF
Degree: Master of Science in Decision Information Systems
School: Arizona State University
Date: 1994
Pages: 38 (text) 18 (code)

Accesion For		
NTIS CRA&I	☒	
DTIC TAB	☐	
Unannounced	☐	
Justification _____		
By _____		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

ABSTRACT

Client/Server Computing is one of the newest technologies in distributed systems. It allows different computers to communicate and share resources. The idea is relatively simple however, the underlying factors make it difficult. This paper explores the use of remote procedure calls (RPC) to create a distributed, client/server application. The presentation includes a discussion of RPC along with a simple application that introduce the reader to RPC. Finally, an application in the area of file migration is presented.

The program is designed to receive a requested file from the user, it then accesses a file to find the location of the requested record and then the record is retrieved. For the purpose of the simulation the file is transferred when it is located in a different place than that of its owner. While this is not a complex file migration algorithm it lends itself to such applications as moving personnel records , dynamically, so that they reside in the same location as their owner in order to facilitate communication and reduce costs. Similarly, another application might involve having medical records follow a person from place to place. It is easy to see that client/server computing has the potential to be powerful and tools such as remote procedure calls are essential to this technology

BIBLIOGRAPHY

Bloomer, John. <u>Power Programming with RPC</u>. Sebastopol: O'Reilly and Associates, Inc., 1991.

Carpenter, B. E. and R. Cailliau. "Experience with Remote Procedure Calls in a Real-time

Control System." <u>Software--</u> <u>Practice and Experience</u>. Vol. 14. September 1984, pp.

901-07.

Comer, Douglas E. and David L. Stevens. <u>Internetworking with</u> <u>TCP/IP</u>. Vol 3.

Inglewood Cliffs: Prentice Hall, Inc., 1993.

Curry, David A. <u>Using C on the UNIX System</u>. Sebastopol: O'Reilly and Associates, Inc., 1989.

Hac, Anna. "A Distributed Algorithm for Performance Improvement Through File Replication,

File Migration, and Process Migration." <u>IEEE Transactions on Software Engineering</u>. Vol

15, No 2. November 1989, pp. 1459-1470.

Hahn, Harley. <u>A Student's Guide to UNIX</u>. New York: McGraw-Hill, 1993.

Hewlett-Packard Company. <u>HP Visual User Environment 3.0 User's Guide</u>. Hewlett-

Packard Company, 1992.

Kernighan, Brian W. and Dennis M. Ritchie. <u>The C Programming Language</u>. 2nd ed. Murray Hill:

AT&T Bell Laboratories, 1988.

Korzeniowski, Paul. "Make Way for Data." <u>Byte</u>. June 1993, pp. 113-115.

Levy, Henry M and Ewan D. Tempero. "Modules, Objects and Distributed Programming:

Issues in RPC and Remote Object Invocation." <u>Software--Practice and Experience</u>.

Vol. 21.January 1991, pp. 77-90.

Soto, J. L. Cruz, M. C. Calzada Canalejo and M. Marin Beltran. "Parallelization of Differential

Problems by Partitioning Method (Synchronized Algorithm)." Computers and

Mathematics with Applications. July 1993, pp. 25-31.

Sun Microsystems. Network Programming Guide. Sun Microsystems, Inc., 1990.

Waite, Mitchell and Stephen Prata. New C Primer Plus. Carmel: Sams Publishing, 1993.

A Demonstration of Client-Server Technology

Using Remote Procedure Calls

For An Applications of File Migration

For Moving Records Based on Location

by

Andrea Miller

An Applied Project Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science in Decision and Information Systems

Arizona State University

December 1, 1994

EXECUTIVE SUMMARY

Client/Server Computing is one of the newest technologies in distributed systems. It allows different computers to communicate and share resources. The idea is relatively simple however, the underlying factors make it difficult. This paper explores the use of remote procedure calls (RPC) to create a distributed, client/server application. The presentation includes a discussion of RPC along with a simple application that introduce the reader to RPC. Finally, an application in the area of file migration is presented.

The program is designed to receive a requested file from the user, it then accesses a file to find the location of the requested record and then the record is retrieved. For the purpose of the simulation the file is transferred when it is located in a different place than that of its owner. While this is not a complex file migration algorithm it lends itself to such applications as moving personnel records , dynamically, so that they reside in the same location as their owner in order to facilitate communication and reduce costs. Similarly, another application might involve having medical records follow a person from place to place. It is easy to see that client/server computing has the potential to be powerful and tools such as remote procedure calls are essential to this technology

TABLE OF CONTENTS

PREFACE

This project represents one part of three related and closely coordinated projects. In total, the three projects provide: a) exposition and programming examples needed to understand the use of Remote Procedure Calling (RPC) for distributed Client/Server processing in networks, b) extensions needed for RPC applications that migrate files among hosts in a network based on several interesting and practical criteria, and c) a discussion and code pertaining to the development of a Graphical User Interface (GUI) for the on-line demonstration of concepts spanning the collection of projects.

Working as a team, Eric DeLange, Andrew Jank and Andrea Miller jointly participated in developing a tutorial discussion of RPC concepts and programming techniques. After the development of fundamental RPC programs which illustrate these techniques, the team collaborated to develop a foundational set of code pertinent to file migration among networked hosts. Thereafter, each team member individually expanded this code to model specific RPC applications of file migration.

Finally, a jointly-developed GUI to facilitate the convenient on-line execution and demonstration of all applications was added. Source files and a discussion of the GUI are of considerable instructional interest because they present techniques for synthesizing window programming and C programs that not only make RPC calls but also make system calls to invoke UNIX scripts and utilities.

For the convenience of the reader, each of the related project reports contain the source code and discussion of the applications developed by the companion authors. This project report

uniquely contains the discussion and source code developed by the author for moving records

based on location. Readers who are interested in time migration or heuristic migration are free to

reference the reports authored by Andrew Jank and Eric DeLange respectively.

INTRODUCTION

With the vast technological advances in computer capabilities, specifically in the distributed environment, organizations are moving away from centralized systems in order to avail themselves of the various advantages offered by distributed systems. Often, organizations are geographically dispersed which leads them to a natural distribution of computing resources. Furthermore, distribution provides a business with enhanced reliability by allowing data replication across multiple sites. Finally, given the size of organizations in today's business arena, a distributed environment facilitates high transaction rates and also allows for the integration of heterogeneous platforms that often accompany mergers between companies or the addition of new locations.

With the advantages of a distributed environment comes the added complexity of communicating among multiple machines across a network. Since the information is no longer on a single machine, there must be a way to track the identities of the requester and the recipient as well as the location of the information required for the transaction. Client/Server technology is an approach which deals with this complexity. In essence, the requesting computer is viewed as a client which asks for a particular service from another machine (the server). When the client requests a service from the server, control is passed to the server until it has completed processing the request, at which time the client receives the requested information and regains control. For example, if a client needs the result of a complex mathematical function, but lacks the processing power to compute it within a reasonable amount of time, it can request the services of another processor (server) that has the necessary capabilities to perform the calculation. Once the

calculation has been performed, the server passes the answer, and control, back to the awaiting client.

The above example looks very much like a function call in any standard language such as C or FORTRAN and, in fact, it is. The only difference is that the function call is made over a network to a different machine. In truth, making a function call to another machine is aptly accomplished through the implementation of Remote Procedure Calls (RPC). RPC is a popular framework for programming in a distributed client/server environment. It provides a means by which a client can communicate with a server.

This project develops an application using RPC. Specifically, it illustrates the usefulness of RPC and its applicability in the area of file migration. In achieving this end, a small, simple program was developed to provide a basic understanding of RPC. Once a working application in the area of file migration was achieved, each member of the group modified the program to accommodate variations in the file migration model. Finally, a front-end was added on the Hewlett-Packard UNIX System to make the program more user-friendly.

REMOTE PROCEDURE CALLS

Remote Procedure Calling permits a client to execute procedures on other networked computers. In fact, RPCs serve as the basis for a majority of the distributed system utilities currently in use (i.e. NFS and NIS). A major reason for RPC utilization is the ease with which RPCs can be implemented, when compared to the lower-level network socket interfaces which have been required prior to the advent of RPCs. Moreover, RPCs are perceived as powerful programming tools, especially for users, since their implementation resembles traditional programming methodologies and the network interfaces are provided with increased transparency to the users (Bloomer 1).

RPC has been identified as a type of middleware. Middleware is software that translates communication between different machines or platforms. This type of software protocol is often necessary for communications within a client/server environment. RPC has been deemed one of the two primary types of client/server middleware; the other is message processing (Korzeniowski 114). In order to use RPC, synchronous links between computers must be established, either using datagram or TCP transports. If no transport is available at the time of initiation, the client application will automatically wait for an answer from the server, and eventually "time out" (halt) when no reply is gathered. Message passing differs from RPC protocol in that messaging systems work on the store and forward principle (Korzeniowski 114). Store and forward systems allow a server to read a computer request message at its own convenience. Since message passing systems do not wait for a response from the server function, these types of systems support asynchronous client/server interaction. However, the synchronous connection supported by RPC

communication maintains a higher degree of reliability than message processing. In a message passing system, for example, it is possible that a message may never be received and neither the source, nor the receiver, would be aware of a problem.

RPCs have many other advantages, beyond the simple benefit of enhanced reliability. Some other advantages include the ability of RPCs to run on hosts having different operating systems, the ease of incorporation of RPC into various software products, and the ability to utilize unused CPU time at distant machines. However, RPC implementation has its drawbacks. RPC lacks flexibility and is often difficult to use with many servers. Message passing offers a greater degree of flexibility, along with an easier programming procedure for establishing asynchronous communication between processes in a networked environment. In summary, message passing lacks the reliability and standards that RPC provides and is limited in use (Korzeniowski 115).

RPCs vs. Local Procedure Calls

A remote procedure call appears extremely similar to a local procedure call--as intended. The difference between the two procedures is that in a local procedure call, the client process initiates a procedure in its own address space, whereas with RPC the server and client exist as two separate processes, usually on different machines (Comer 289). It is this separation of processes that allows the server function to reside on a different machine. Nevertheless, it is important to note that RPC can still be utilized when the client and the server execute on the same machine (Comer 306).

During normal implementation of RPC, the client process and the server process communicate to each other via two stubs (Levy 79), namely the client stub and the server stub. A stub is a communications interface that establishes the RPC protocol and determines how each message is constructed by the processes and interchanged between the two. The client process first consults its own stub to locate any remote processes that are required for program operation. Subsequently, the client makes the necessary requests of those processes. Meanwhile, the server (daemon) perpetually listens to the network, through the server stub, for any requests transmitted by clients. More specifically, one daemon, the Inetd, serves as a "grandfather" daemon for all other daemons. Inetd runs perpetually and starts other server daemons upon receipt of requests for server services. The server fulfills each request in succession, returning to its waiting state after completion of each request (Bloom 2).

At a more basic level, each server process is identified by a port (logical network communication channel) by which it establishes communications for client requests. When a server is initiated on a machine, the computer establishes an address (port) for server communications. This address, which is unique, is registered with the server machine's portmapper (Bloom 11-12).

The portmapper itself provides a crucial network service for all client/server communication. Its job is to keep track of all services that are available on a machine and their port addresses. Whenever a client requests a service from a particular machine, the client petitions the portmapper for the service. If the requested server exists, the portmapper establishes a communication channel between the client and the server. Even when a client and server reside

on the same computer, the operation for establishing a link between the client and the server is the

same; the network is still involved in the communication. The client still checks with its stub for

the server's address, only in this case the address provided by the portmapper would correspond

to the same machine. In effect, the request travels across the network only to return to its origin

(Sun MicroSystems 36).

A Foundational Client/Server Demonstration

To facilitate the comprehension of programs upon approaching the tasks entailing RPC, it

is helpful to begin with a straight-forward application. For this purpose, a client/server RPC

demonstration has been developed wherein a client process passes an integer to a server that

increments the integer and returns the updated value to the client. The files necessary to perform

this, and any other, RPC application include a protocol definition file, client program, server

program, and the stubs and header file generated by the RPC compiler after these are created.

Before programming the client and server processes, it is first necessary to create a

protocol definition in the remote procedure call language (RPCL). A protocol definition is a file

that describes both the list of data structures that will be passed between the client and server, and

the function call required from the client in order to use the server's resources. The initial

protocol definition file, *add_it.x*, can be referenced in Appendix A.

The critical elements in the protocol definition include a unique program number and

version numbers within each program. In this example, the program is called ADDPROG and is

assigned the unique number 0x20000002. It is imperative that this value be unique since it is used

by the portmapper to identify the process (from poignant experience, the author can readily attest

to the confusion that results from having multiple programs with the same program number). The

version numbers are useful when updates warrant the need for a distinction between the original

version, and the subsequent update (Bloomer 43). For example, the *add_it* protocol definition file

contains one version with a simple function definition called ADD_NUM which receives and

returns an integer. Another version could be defined (identified by the number 2) which passes a

structure instead of an integer. RPCL is similar to the C language, though this simple example

does not make this apparent; however, the protocol definition file for the author's application

(Appendix B) demonstrates the similarity between declared constants, structure definitions, and

additional functions.

Once the user has created the protocol definition file, a UNIX program, called RPCGEN,

compiles the definition and produces several files. RPCGEN creates both the client and server

stubs, as well as a header file that defines the RPC parameters that are included in both the server

and client routines. After compilation of the RPC definition file, the next task entails developing

the client and server code.

In the *add_it* example, the client code is labeled *add_it.c*. The code is written in C and

contains familiar formats like *include* statements, variable declarations, etc. Additionally, the

code includes features that are unique to RPC. First, the rpc/rpc.h library must be included along

with the header file add_it.h which is produced by the compiler rpcgen. Next, a special pointer of

type CLIENT is declared which points to a structure that contains information about the port and

socket addresses (Bloomer 7). The value of that pointer is determined by the function clnt_create

which establishes a connection between the server and client machines. This function requires the name of the host with which to establish a connection (can be the same), program name, version name, and the type of transport protocol (tcp or udp). If no connection can be established, the function clnt_pcreateerror is called to inform the user that no connection could be made to the host (Sun MicroSystems 45). Finally, the function that was declared in the protocol definition file (*add_it.x*) is called (the version number is appended to the function name by rpcgen). Passed to the function are the integer which will be manipulated and the CLIENT pointer which contains the communication information.

The last component of the *add_it* example is the server code, which is also coded in the C language. Again, the rpc/rpc.h library is included as well as the header file add_it.h. Noticeably different from the client code is the lack of a main declaration. In essence, the server code is simply a function declaration and can be considered as a function within the client code, only residing on a different machine. Furthermore, all communication between the client and the server is accomplished through pointers. Thus, the client passed the pointer to the integer *num* and the server returns a pointer to the new number which is, ironically, called *oldnum*. The descriptive "Hello People" statement was included to provide optional output to confirm that the server was responding (debugging tool).

Advanced RPCs

Once this simple RPC application that adds two integers over a network was completed, the level of difficulty was increased by working with strings and structures, until the author was versed enough in RPC protocol and application specification to begin work on the file migration

application itself. It uses many of the fundamental concepts of the simpler RPC application, as well as additional, more complex concepts. A copy of the code is included in Appendices B through E. Appendix B is the protocol definition file, Appendix C contains the client code, Appendix D contains the server code, and Appendix E has all of the scripts written in UNIX that are used through system calls by the client and server code as well as the protocol definition file and server code for the second server.

The application of file migration, of course, is not the only area where RPC can be employed effectively. There are many other applications which lend themselves to the advantages of RPC, including using RPC to calculate partial derivatives. In this application, the equations are partitioned and numerically solved on different computers. The fragmented solutions are then pooled to obtain the final result (Soto 25-27). Another application area encompasses using RPC as a tool for software applications dealing with real-time process control systems that are large and complex (Carpenter 901-902). Undoubtedly, the use of RPC will become more widespread and, as it does, we will see an increase in the number of applications in this area.

FILE MIGRATION

Many organizations have expanded significantly, in a physical and geographical sense, over the past few decades. This growth has been accelerated by the fast-paced nature of the advancing computing environment. Many organizations have noticed that it can be beneficial to modify the location of various frequently accessed company files from their current locations to others. By varying the location of files appropriately, communication costs and file transfer duration times, which typically result from locating and acquiring large files (such as company accounting histories or files containing multiple graphic bitmaps), can be minimized. Companies have often solved these problems by manually modifying the primary location of particular files, or by replicating the files to multiple locations.

A Case for Computer-integrated File Migration

For example, most world-wide corporations maintain personnel files for each and every member of their company. Regardless of each member's current location, it is imperative that the data contained within each file remains current. In order to maintain currency, the file must be updated continually as to reflect each person's current location, position, job status, possible job qualifications, personal preferences, etc. Most companies maintain each member's personal record at one specific site. The file is usually located closest to where that person normally conducts business (at some home-base location). Whenever their particular personnel file must be updated, their central file must be found and modified. Although this method may seem prudent, as long as the person remains at that specific location for an extended period of time and rarely

deviates far from that location on business ventures, the reality of global business dictates that many personnel are frequently dispatched world-wide for extended periods of time at irregular intervals.

If a person is temporarily reassigned from one business site to another, for example, as the schedule of a sales representative could require, maintaining a file at a central location may be considered unwise. It makes sense, then, to have each company member's personal file "follow" them to their current business location, as required by their occupation. Many organizations have implemented this concept by manually moving personnel files from one location to another, as warranted. The process of moving this file is usually performed manually, but why bother when a computer system can perform the necessary file transfers, at close to optimal times, and minimize transferal and access costs concurrently? In most cases, the manual process of moving files is either cost-ineffective, inefficient, or subject to oversight errors.

A case for Computer-integrated File Replication

Related to the prior example, a person may perform certain business functions at two or more distant locations on a frequent and extended basis. In order to minimize communication costs, their personal file could be manually moved between these locations as their business functions require. However, this method may not be the most efficient and cost-effective one for determining the timing of file relocations. Routinely moving a file among two or more locations seems like a senseless task. If the employee regularly returns to a limited set of specific locations, having information specialists repeatedly relocate the member's personal file to each of these

locations could result in unnecessary file transfer costs. If this person's file is only needed at each

location for read-only applications, a simple file copy at each location would minimize the cost of

access times and would require only one file transfer for each location. Any competent computer

specialist could determine when a file should be copied to another location, but why employ one if

the computer system can perform this function on its own? The system could apply a set of

criteria and automatically determine when a file should be copied to another location. The

possibility of automatically moving files to the optimal locations leads to the concept of file

migration.

The Benefits of Computer-integrated File Migration

The two preceding examples illustrate the advantages of computer-integrated file

migration. Although many methods can be employed to determine when a file should be either

replicated or moved, there is little evidence to support that the file transfers themselves should be

performed manually by the computer system monitors. A computer routine can be developed that

determines the optimal distribution of each file and performs the necessary relocation(s) without

human involvement.

The concept of file migration is, more or less, an automatic process. In this process, the

organization pre-defines a specific set of criteria that must be met before a file is either relocated

or replicated to another site. When the pre-specified criteria are met, the awaiting computer

functions perform the desired file relocation and replication without any interaction by the

computer system monitor. In this scenario, a centralized database would maintain the criteria as

well as current information on the location of each file, the type of each file, and the types of accesses available to each file (e.g. if a file is read-only, write-only, or both). If the file location(s) have been modified, only a simple change to the database is required.

Although it may seem expensive to require each file query to consult the central database prior to any, and every, file access, the cost will usually be negligent, especially when compared to normal file transfers. For example, a personnel file that contains complex images (including dental exam x-rays, photographs, and other bitmaps) can often take hours to transfer error-free over a large distance, while a simple database call to another machine takes much less than a few seconds.

Analysis of Computer-integrated File Migration

So, as far as the added communication overhead is concerned, file migration adds little to the overall cost. Actually, this extra overhead cost is required from all personnel database systems, since any distributed system requires the existence of, and access to, some type of database to determine a particular file's location. The real savings from the file migration concept comes from the efficiency gained by the elimination of human interaction at the file transfer level.

The transferal/replication criteria are only specified at a single point, with periodic updates, eliminating the necessity of human decision-makers to determine the location of each file at each point in time. The system itself takes care of the implementation issue.

THREE APPLICATIONS FOR FILE MIGRATION

In order to demonstrate RPC, three different, but related, file migration applications were chosen with regard to unit personnel records. For instance, in the Air Force, currently all records are kept on one database in one location. While this may seem like the simplest way to maintain a database, it might be more cost-effective to distribute the files and move them electronically. This is the basic idea behind the applications. While each option is a slightly different scenario with respect to migrating personnel records, all three applications have an underlying goal as well as a communal set of code.

This shared objective is to create an application that will maintain databases of the actual personnel records along with a database that knows the location of each record. Furthermore, each application will display a set of statistics regarding the location and number of accesses of a particular file from a particular location. This enables an individual to keep track of where the file is used and requested most often. Finally, based on some predetermined set of rules (here is where the different options are derived) the file is transferred to a different location and a message appears telling the user that the file has been transferred to a new location. The transfer is also updated in the database that tracks the locations of the files.

As mentioned before, the file transfer is governed by a set of rules. Here is where each application finds its identity. The first option deals with transferring a file based on a location that is different from the current location of the file. The second option transfers a file based on the percentage of requests made. The final option transfers a file at regularly scheduled times to given locations. There exist complex algorithms that dictate when a file should be migrated based

on a system's resource utilization, file sizes, and the number of write and read accesses to those files (Hac 1459); however, because the emphasis of this paper is on remote procedure calls rather than file migration techniques, these algorithms are beyond the scope of this project.

Shared Application Features

Though each of the three applications address a different decision rule, they all share certain attributes, processes, and implementation techniques. Since the authors developed a generic set of code around which the applications are built, each application shares a number of the same variables. Furthermore, many of the functions within the server code are identical since the same file information is being processed in each application.

Of special interest is the use of the rdbpt server as a client of the fts server. In other words, the same process can function as a server and a client! The server process need only make a request of another server to become a client, too. This feature in the code required extensive debugging for two reasons: first, the location in the code of the second server call was important to the server's operation and second, the structure of the compilation file was altered significantly in order to integrate the additional stubs into the original configuration.

Finally, each application implements C's capability to make UNIX system calls. This capability allows the integration of UNIX scripts which are triggered by C with a simple system call to the name of the script. In this way, the various tasks of the applications are performed in the most appropriate environment.

For example, it is easier to work with files using UNIX rather than C since the code resides on

UNIX machines. The UNIX system calls are easily identified throughout the code.

THE APPLICATION

Keeping track of personnel records for a large organization can be very cumbersome. If the organization had the ability to let the computer keep track of the location of each person's records and distribute them in a way that minimizes costs, such as communication and storage costs, they would be eager to use it. This RPC application deals with migrating files based on the owner's current location.

There are many potentially interesting applications of file migration based on the location of the owner and file. One application is having medical records follow a patient from one medical center to another, this would be especially favorable if all records are stored in computer databases. Another application is for storing social security records near the location of the individual that they belong to. These records are frequently accessed, thus locating them with the individual would reduce communication costs. Academic institutions could benefit by automatically receiving copies of transcripts from previous schools that a student has attended because the transcripts would electronically follow the student. One final application would involve Email or voice mail addresses following the user from site to site. This would make it possible for individuals to need only one address in a lifetime. These are only a few possibilities the reader is free to explore other possibilities.

In order to make this application work, several pieces of information must be kept track of by the computer. First, it must have the information contained in the actual file whether it be a person's medical records or mail. Next, there must be a database that knows the location of that file, and finally another database must know the location of the owner of the file. A daemon

could then be invoked that periodically checks the two locations and if they are different then the file is transferred to the location of the owner. However, this is similar to the application done by Andrew Jank (reference Andrew Jank's Project). For the purpose of this project all requests for a particular file must be made to a central location, that of the location of the owner.

The crucial assumption for this model is that the requesting server is located in the same location as that of the individual. Therefore, all access to an individual's records is made via their "home office". Even if an individual does much of his business out of town, he is generally assigned one office as his primary place of employment (the "home office" can apply to any of the above situations and is being used here in one situation for ease of reading). Thus, based on the assumption, if someone other than the home office wants access to an individual's records, that request must be made to the individual's home office and they will get the information or perform the necessary transactions. While this may be a big assumption, it is likely that some organizations will find this to their liking. The basic reason is that organizations like to maintain control. Upper management, in particular, does not like the idea of distributing data because it is then no longer necessarily under their control. By the same token, by requiring that outside requests come through the home office, control is maintained over the files at its site. This assumption makes it easier for the file migration routine to know when to transfer a file.

Since the assumption is that the requesting server is located in the same place as the individual, and because all requests are made via that server, to determine if the file is in the correct location, the file location can simply be compared to the location of the requesting server. If they are different the file should be transferred to the same location as the owner. This is

exactly what the first application does. It transfers files if the requesting location is different from where the owner and file currently reside. Once the file has been moved, the database containing the directory of where all the files are located is also updated.

The first application simply moves files based on location. It does not take into account the number of accesses from a different location or calculate any costs or ratios. In this respect, this option may be lacking. Because location is the only decision factor, it may overlook many other factors. Two ensuing applications attempt to address these additional factors (Reference papers by Andrew Jank and Eric DeLange).

Application Features

In order to present some of the features of the code it is first necessary to describe more specifically what the application does. When the application is initially run two windows will pop up on the screen. One window is the file transfer window the other is the active window. The file transfer window displays transfer status (see figure 1). It informs the user of files that have been transferred or that a file will remain where it currently resides. The active window has two functions (see figure 2). The first is that it receives requests for a particular file. The requested file is found and then displayed along with statistical information regarding how many requests have been made from a particular server. In this application, the statistical information is simply additional information. It could be used in more complex file migration algorithms in the future.
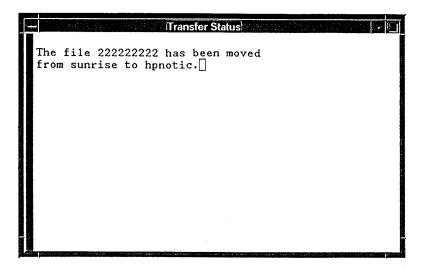
```
┌─ ─────────────────── Transfer Status ──────────────── · ─┐
│ The file 222222222 has been moved                        │
│ from sunrise to hpnotic.□                                │
│                                                          │
│                                                          │
│                                                          │
└──────────────────────────────────────────────────────────┘
```

figure 1

```
┌─ ─────────────────────── PROJECT ONE ─────────────────── · ─┐
│                                                             │
│  ─────────────────────────────────────────────────────     │
│      SSN              Name              Address             │
│  ─────────────────────────────────────────────────────     │
│                                                             │
│   222222222        REED.HANSCOM       19898.GRANDVIEW.DRIVE │
│                                                             │
│                                                             │
│   Sunrise          Sunburn        Sunspot        Hpnotic    │
│      0                0               0              1       │
│  ─────────────────────────────────────────────────────     │
│  ─────────────────────────────────────────────────────     │
│                                                             │
│ andreami@hpnotic:24 □                                       │
│                                                             │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

figure 2

Protocol Definition File

Now that the reader is more familiar with the functionality of the code it is possible to highlight some of the areas of interest in its design. First the protocol definition file is in itself unique because most computer science programs require nothing similar to it. For this application the records are defined as structures (*rdbpt.x* 15-20, 25-33). In some cases it is easier to pass an entire structure back and forth over a network than to individually pass data. The type "string" is also something not found in a C program. In RPC this type is only defined in the protocol definition file. Finally, for this particular application only two functions make requests to the server (*rdbpt.x* 50-51). As earlier mentioned, each RPC application must have code for both the client and the server. Highlights of the client code will be discussed next.

Client Code

The client is the process that makes requests from the server to complete its task and it also establishes the connection to the server. The connection function is predefined and requires four arguments: hostname, program name, program version, and transport type. To see its use see rdbpt.c lines 56-61. Once the connection has been made, the client can then make requests to the server. In this application two procedures, ssn_key and get_rec are requested (ln 64 and 67 respectively). The server will execute the procedure and first send back the location of the requested file and then will retrieve the file itself. Once that information has been returned to the client, it then has the job of presenting it. In this application the formatting and printing of the

information (the requested record and statistics) are done in the client (*rdbpt.c* 73-87). While the

client makes the requests and formats the output it is the server that does the majority of the

work.

Server Code

In this application the server code reads the database and finds the location of the

requested record, along with the record itself. It also keeps track of file location updates and

statistics regarding the number of accesses from a particular location. Finally, the server in this

application also functions as the client process for another server. Lines 30 though 43 of the

server code show the procedure for reading the central database that keeps a record of where

certain files are located as well as, statistics regarding the number of accesses from a particular

location. First space must be allocated for the data and then the information can be read.

Counters are used to keep track of the number of accesses from each location (*rdbpt_svc_proc.c*

57-64). Once the information has been read a comparison is made that determines if the file

resides in the same location as the owner (rdbpt_svc_proc.c 66-72). For simulation purposes it is

also here that the file location is changed. For this application the file is not actually moved

anywhere. The migration is simulated by changing the location of the file in the central database.

Once the data has been read and modified it can be written back to the database

(rdbpt_svc_proc.c 73-80). A final highlight of the server code is the code that allows the server

to be a client for another server process (rdbpt_svc_proc.c 157-170). In this application the

requested file along with the old and new locations (possibly the same) are passed to another

server process. While for the purpose of this simulation the second server process only serves as

an informer, it is here that the file migration routines could actually be invoked and the file would

actually be moved. One element that is essential to RPC that can be seen both in the server and in

the client code is that all variable that are passed between processes are referenced as pointers

(rdbpt_svc_proc.c 17-22). The protocol definition file and server code for the additional server

(*fts.x and fts_svc_proc.*) are located in appendix E.

This discussion is not all encompassing however it does show some of the more important

and interesting elements of the program.

RPC FRONT ENDS WITH HP-UX

The front-end display is an important addition to any application, especially for users. Applications that use RPC technology typically produce a large number of complicated, and almost indecipherable, files. Additionally, in order to run any RPC application, the server application must already be waiting for a request at the server end, before the client can be executed. The typical user does not know, or care, about any of the implementation issues. One attractive quality of client/server computing is that the complicated aspects of computer integration and application are transparent to the user. Therefore, it is important that any user interface that utilizes RPC functions facilitates running the program, and all of its additional files, without involving the end-user in the details of its underlying operation.

The HP-UX Workspace Environment

The HP-UX machines provide several different viewing sessions that offer applications and tools that can be accessed by simply clicking a pointing device (ex. a mouse) on an icon. Two of these sessions are HP VUE and HP VUE lite. The basic differences between the two are that the HP VUE lite session has a front panel with different features and different methods for determining how the view manager interprets the information provided by the user. These panels are typically displayed whenever the user logs on, much how Microsoft Windows provides a set of panels, each of which contains a set of program icons. Users can customize their own panels, but only within the limits set by the System Administrator, who determines what type of options are available to each class of user.

Developing a Control

The application was developed within the HP VUE session. The HP VUE session panel

consists of a top and bottom row. In order to modify these rows several changes must be

performed: an icon must be constructed, an action that will be associated with that icon must be

defined, the action must then be linked to the icon to define a control, and finally the control (the

icon with an underlying action) must be added to the front panel by editing the ".vuemrc" file.

Building Icons

Creating an icon is relatively easy for anyone who has a creative mind and likes to draw.

The IconEditor is specifically designed for this purpose; it provides the user with a drawing

screen and tools to construct various icons. Once the icon is finished, the constructor need only

save it for future use. In order to have the icon associated with an application, an action must be

specified and attached to the icon.

Creating Actions

Creating an action is accomplished by simply using the CreateAction icon in the HP-UX

General Toolbox. The user inputs the action name, a command line to initiate the application, and

a type of window to display the action results within. An action can only be specified once; if

future modifications must be made, a new icon-action definition must be specified. Once the icon

and action associated with it have been created, the newly defined control can be added to the

panel.

Adding a Control to the HP-UX VUE Session

The ".vuemrc" file must be edited to include the new control. The ".vuemrc" file contains all the control definitions, as well as the names for the controls that reside in the top and bottom rows of the panel. These controls can take on different actions, as well as combinations of behaviors including: push button, drop zone, file monitor, client window, and toggle button.

To include a new control to the ".vuemrc" file, several steps must be taken. Initially, the icon controls are placed in the appropriate box and are then defined. The box placement describes in what order the various controls will be displayed. The word "CONTROL" along with the name for a new control are included among the other control definitions for a desired box (top or bottom) and position within the box. The type of control must also be defined. In the case of the application, a button "TYPE" control is used to initiate the application. For any control addition, the "PUSH_ACTION" for the button must also be specified. The "PUSH_ACTION" can either be an action similar to the one created above, or an executable command. An action similar to the one created above is incorporated into the author's RPC application. Finally, an "IMAGE" must be specified for each control. The image is simply the icon name. Little more is involved with modifying the view configuration of the HP-UX

machines to include any, and all, new applications to the window manager.

Restarting the Workspace

In order to incorporate the new controls to the window manager, all one needs to do is restart the workspace manager. The new control definitions will take effect and function in all

future uses of the start-up window manager workspace. By using icons and associating them with actions it is possible to execute the RPC application in an environment that not only preserves the benefits of client/server computing, but also retains the invaluable element of end-user transparency by isolating the user from the underlying complications involved with RPC technology.

CONCLUSIONS

Though the author's experience in the area of RPC is limited, by researching the subject and working on a project that incorporates its capabilities, the usefulness of RPC in communicating in a client/server environment was recognized. Moreover, even though the project was somewhat limited, the complexity involved in implementing client/server technology is readily discerned. Of course, the project was not completed without learning something, nor were the possible applications of RPC in the realm of file migration exhausted.

Lessons Learned

As with any project, there is no substitute for the lessons one learns from experience. This project is no exception. First and foremost, it cannot be stressed enough how important it is that the program number defined in the protocol definition file be unique. If there are two programs with identical values, a clear communication channel cannot be established between a client and **one** server since, according to a client's stub, two server's will exist for the same purpose.

Another important issue in dealing with RPC is passing parameters. It is necessary that all variables be passed as pointers for RPC to work properly. Additionally, the value that is to be returned from the server function must be declared as a static variable.

<u>Future Endeavors</u>

In order to keep the scope of this project within reason, the three file migration applications were approached from a simulation perspective. That is, no physical migration of files actually occurs. When a file is said to have been moved from one host to another, the only thing that has changed is the database file that would track the migration of files. Obviously, this leaves open the possibility of expanding the project by physically migrating the files. Furthermore, additional applications can be developed such as maintaining duplicated files at separate sites (read-write and read-read access are important in this area). Along with this, more research can be done in adequate rules that define at what point a file is to be moved or duplicated. This does not comprise an exhaustive list of supplementary application areas to the project and the reader is free to explore their own possibilities.

BIBLIOGRAPHY

Bloomer, John. <u>Power Programming with RPC</u>. Sebastopol: O'Reilly and Associates, Inc., 1991.

Carpenter, B. E. and R. Cailliau. "Experience with Remote Procedure Calls in a Real-time

Control System." <u>Software-- Practice and Experience</u>. Vol. 14. September 1984, pp.

901-07.

Comer, Douglas E. and David L. Stevens. <u>Internetworking with   TCP/IP</u>. Vol 3.

Inglewood Cliffs: Prentice Hall, Inc., 1993.

Curry, David A. <u>Using C on the UNIX System</u>. Sebastopol: O'Reilly and Associates, Inc., 1989.

Hac, Anna. "A Distributed Algorithm for Performance Improvement Through File Replication,

File Migration, and Process Migration." <u>IEEE Transactions on Software Engineering</u>. Vol

15, No 2. November 1989, pp. 1459-1470.

Hahn, Harley. <u>A Student's Guide to UNIX</u>. New York: McGraw-Hill, 1993.

Hewlett-Packard Company. <u>HP Visual User Environment 3.0 User's Guide</u>. Hewlett-

Packard Company, 1992.

Kernighan, Brian W. and Dennis M. Ritchie. <u>The C Programming Language</u>. 2nd ed. Murray Hill:

AT&T Bell Laboratories, 1988.

Korzeniowski, Paul. "Make Way for Data." <u>Byte</u>. June 1993, pp. 113-115.

Levy, Henry M and Ewan D. Tempero. "Modules, Objects and Distributed Programming:

Issues in RPC and Remote Object   Invocation." <u>Software--Practice and Experience</u>.

Vol.   21. January 1991, pp. 77-90.

Soto, J. L. Cruz, M. C. Calzada Canalejo and M. Marin Beltran. "Parallelization of Differential

Problems by Partitioning Method (Synchronized Algorithm)." Computers and

Mathematics with Applications. July 1993, pp. 25-31.

Sun Microsystems. Network Programming Guide. Sun Microsystems, Inc., 1990.

Waite, Mitchell and Stephen Prata. New C Primer Plus. Carmel: Sams Publishing, 1993.

APPENDICES

Appendix

A

```
1    /*   ADD_IT.X    * /

2    /*   THIS  FILE  IS  USED  BY  RPCGEN  TO  PRODUCE  THE  HEADER  FILE  ADD_IT.H  * /



3    program  ADDPROG  {
4            version  ADDVERS  {
5                    int  ADD_NUM(int)  =  1;
6            }  =  1;
7    }  =  0x20009300;
```

```
1    /*      ADD_IT.C    */

2    /* THIS IS THE CLIENT PROCEDURE */


3    #include <stdio.h>
4    #include <ctype.h>
5    #include <rpc /rpc.h>
6    #include "add_it.h"

7    main()
8    {
9    CLIENT *cl;
10   int num;
11   char *hostname[20];


12   printf("\nPlease enter the remote host name: ");
13   gets(hostname);

14   printf("\n\nPlease enter the number to increment: ");
15   scanf("%d", &num);


16   /* THIS IS THE CLIENT HANDLE THAT ESTABLISHES THE CONNECTION */
17   /* BETWEEN THE CLIENT AND THE SERVER */


18   if (!(cl = clnt_create(hostname, ADDPROG, ADDVERS, "tcp")))
19   {
20      clnt_pcreateerror(hostname);
21      exit(1);
22   }

23                                       /* HERE IS THE PROCEDURE CALL */

24   printf("\n\nThe new number is %d\n", *(add_num_1(&num, cl)));

25   }
```

```
1    /* ADD_IT_SVC_PROC.C */

2    /* THIS IS THE SERVICE PROCEDURE */



3    #include <stdio.h>
4    #include <string.h>
5    #include <rpc /rpc.h>
6    #include "add_it.h"


7    int *add_num_1(oldnum)
8    int *oldnum;

9    /* THIS FUNCTION ADDS 21 TO THE INPUT NUMBER AND THEN RETURNS THE NUMBER */

10   {
11       printf("\nHello People");
12       *(oldnum) += 21;
13       return oldnum;

14   }
```

Appendix

B

```
 1    /*                    *** project 1 ***                                    *
 2    /*                                                                         *
 3    /*                         RDBPT.X                                         *
 4    /*                                                                         *
 5    /*              This program is used by RPCGEN to produce the header       *
 6    /*          file for the RDBPT RPC.                                        *
 7    /*                                                                         *


 8    const MAX_REC_LEN = 255;           /* max personal record length
 9    const SSN_SIZE = 9;                /* size of Social Security Number
10    const NAME_SIZE = 80;              /* size of person's name
11    const ADDR_SIZE = 80;              /* size of person's address

12    const HOST_SIZE = 255;             /* size of host computer


13    /*                                        Defines the structure of each personal    *
14    /*                                        record: SSN, name, and address            *


15    struct pers_rec
16    {
17            string      ssn<SSN_SIZE>;
18            string      name<NAME_SIZE>;
19            string      address<ADDR_SIZE>;
20    };


21    /*                                        Defines the structure of the database     *
22    /*                                        record for each file: ssn (filename),     *
23    /*                                        current location, 4 counters to measure   *
24    /*                                        file accesses from each server.           *


25    struct dbase_rec
26    {
27            string      ssn<SSN_SIZE>;
28            string      loc<HOST_SIZE>;
29            int         sunrise;
30            int         sunburn;
31            int         sunspot;
32            int         hpnotic;
33    };


34    /*                                        Defines the structure of an information    *
35    /*                                        record about a specific file, including    *
36    /*                                        the SSN (filename), the local host from     *
37    /*                                        the file has been requested, and where      *
38    /*                                        the file is actually located.              *


39    struct inputrec
40    {
41            string ssn<SSN_SIZE>;
42            string lc_host<HOST_SIZE>;
43            string h_name<HOST_SIZE>;
```

```
44    };
45



46    program  RDBPT_PROG
47    {
48            version  RDBPT_VERS
49            {
50                    dbase_rec  SSN_KEY(inputrec)  =  1;
51                    pers_rec  GET_REC(inputrec)  =  2;
52            }  =  1;
53    }  =  0x20009302;
```

Appendix

C

```
 1    /*                   *** project 1 ***                          * /
 2    /*                                                              * /
 3    /*                         RDBPT.C                              * /
 4    /*                                                              * /
 5    /*              This is the client code for the RDBPT service.  * /
 6    /*    The client receives as input the hostname and the file-   * /
 7    /*    name for access.   The code translates these inputs into  * /
 8    /*    the proper form, contacts the RDBPT server to find the    * /
 9    /*    actual location of the file, then retrieves and prints    * /
10    /*    the file for the user.                                    * /
11    /*                                                              * /


12    #include <stdio.h>
13    #include <string.h>
14    #include <rpc /rpc.h>
15    #include "rdbpt.h"
16    #include <stdlib.h>



17    main (argc, argv)
18              int               argc;
19              char              *argv[];


20    {
21              CLIENT            *cl;                /* Client  handle
22         dbase_rec         *d_record;           /* Retrieved  dbase  record
23              pers_rec          *p_record;         /* Retrieved  personal  record
24         inputrec          inrec;            /* Information  record  (ARGV's)
25              FILE              *c_fp = NULL;       /* FP  for  hostname  conversion
26         FILE              *ssn_p = NULL;      /* FP  for  SSN  sonversion
27              char              HSTNAME[HOST_SIZE];
28                                                /* Temp  variable  for  hostname



29    /*                                    Assign  filename  and  hostname  to   *
30    /*                                    variables  for  use                   *

31              inrec.ssn  = argv[2];
32              inrec.h_name  = argv[1];



33    /*                                    Convert  hostname  to  proper        *
34    /*                                    representation                        *

35              system ("find_hname");
36              c_fp = fopen("hname.txt","r");
37              fscanf (c_fp, "%s", HSTNAME);
38              fclose (c_fp);
39              inrec.lc_host = HSTNAME;




40    /*                                    Convert  SSN  filename  to  proper    *
41    /*                                    representation                        *

42              ssn_p = fopen("ssn.txt", "w");
43              fprintf(ssn_p, "%s", inrec.ssn);
44              fclose(ssn_p);
45              system("ssn_ident");
```

```
46            p_record                    = (char *)        malloc(sizeof(pers_rec));
47            d_record                    = (char *)        malloc(sizeof(dbase_rec));


48    /*                                            Provide error usage message            *
49            if ((argc != 3))
50            {
51                    fprintf(stderr, "\nUsage: %s local.server SSN\n", argv[0]);
52                    exit(1);
53            }


54    /*                                            Establish connection to the server     *
55    /*                                            (RDBT) process                         *

56            if (!(cl = clnt_create(argv[1], RDBPT_PROG,
57                                    RDBPT_VERS, "tcp")))
58            {
59                    clnt_pcreateerror(argv[1]);
60                    exit(1);
61            }


62    /*                                            Retrieve file location and access      *
63    /*                                            information                            *

64            d_record = ssn_key_1 (&inrec, cl);

65            strcpy(inrec.h_name,d_record->loc);


66    /*                                            Retrieve actual file                   *

67            p_record = get_rec_1 (&inrec, cl);


68    /*                                            Print out the contents of the file, the    *
69    /*                                            current file location, and the current     *
70    /*                                            file access statistics.                    *

71            system ("clear");

72            printf ("\n\n ——————————————————————— ");
73            printf ("————————————————————\n");
74            printf ("\t  SSN\t            Name\t\t    Address\n");
75            printf (" ——————————————————————————— ");
76            printf ("————————————————————\n");

77            printf ("\n\t%s\t%s\t\t%s\n\n\n\n", p_record->ssn, p_record->name,
78                                            p_record->address);
```

```
79              printf ("\tSunrise\t\tSunburn\t\tSunspot\t\tHpnotic\n");
80              printf ("\n\t      %d\t\t      %d\t\t      %d\t\t      %d\n\n",
81                          d_record->sunrise, d_record->sunburn, d_record->sunspot,
82                          d_record->hpnotic);


83              printf ("     ———————————————————————————");
84              printf ("———————————————————————————\n");
85              printf ("     ———————————————————————————");
86              printf ("———————————————————————————\n\n\n");
87      }
```

Appendix

D

```
 1    /*                  *** project 1 ***                                    *
 2    /*                                                                       *
 3    /*                      RDBPT_SVC_PROC.C                                  *
 4    /*                                                                       *
 5    /*              This is the program that provides services to the        *
 6    /*      RDBPT clients.   The server has two main server functions: one   *
 7    /*      that contacts the central database and retrieves statistical     *
 8    /*      and location information about the desired file, and another     *
 9    /*      that actually retrieves the contents of the personal file.       *
10    /*                                                                       *


11    #include <stdio.h>
12    #include <string.h>
13    #include <rpc /rpc.h>
14    #include "rdbpt.h"
15    #include "fts.h"
16    #include <ctype.h>


17    FILE                *fp1  = NULL;        /* FP for accessing central database
18    FILE                *fp2  = NULL;        /* FP for retrieving personnel record
19    FILE                *ssn_p = NULL;       /* FP for opening temporary files
20    static pers_rec     *p_rec = NULL;       /* pointer for personnel record
21    static dbase_rec *d_rec = NULL;          /* pointer for database record
22    char                *oldname[30];        /* temp variable for old file location


23    /*                         READ_DBASE                                    *
24    /*                                                                       *
25    /*      This function allocates internal memory and reads the central    *
26    /*      database, retrieving the statistics and host information about    *
27    /*      the desired personnel file.                                      *


28    int read_dbase(inrec)
29              inputrec            *inrec;

30    {
31            if (!d_rec)
32            {
33                    d_rec = (dbase_rec *) malloc(sizeof(dbase_rec));
34                    d_rec->ssn = (char *) malloc(sizeof(SSN_SIZE));
35                    d_rec->loc = (char *) malloc(sizeof(HOST_SIZE));
36                    d_rec->sunrise = (int) malloc(sizeof(int));
37                    d_rec->sunburn = (int) malloc(sizeof(int));
38                    d_rec->sunspot = (int) malloc(sizeof(int));
39                    d_rec->hpnotic = (int) malloc(sizeof(int));
40            }

41            if (fscanf(fp1, "%s %s %d %d %d %d",
42                    d_rec->ssn, d_rec->loc,
43                    &d_rec->sunrise, &d_rec->sunburn, &d_rec->sunspot,
44                    &d_rec->hpnotic) != 6)
45                            return (0);

46            return (1);
```

```
47    }



48    /*                              WRITE_DBASE                              *
49    /*                                                                       *
50    /*              This function modifies the statistic counter and file location  *
51    /*              information and updates the applicable central database record. * /



52    int write_dbase(inrec)
53              inputrec              *inrec;
54    {

55    /*                                      Increment  appropriate  counter  if  same  as  the   *
56    /*                                      local  host  requesting  service.                      *

57              if (strcmp(inrec->lc_host, "sunrise")  ==  0)
58                      d_rec->sunrise  +=  1;
59              if (strcmp(inrec->lc_host, "sunburn")  ==  0)
60                      d_rec->sunburn  +=  1;
61              if (strcmp(inrec->lc_host, "sunspot")  ==  0)
62                      d_rec->sunspot  +=  1;
63              if (strcmp(inrec->lc_host, "hpnotic")  ==  0)
64                      d_rec->hpnotic  +=  1;


65    /*                                      Modify  new  file  location                           *



66              system  ("clear");
67              strcpy(oldname,  d_rec->loc);

68              if (strcmp(inrec->lc_host,  d_rec->loc))
69              {
70                      strcpy  (oldname,  d_rec->loc);
71                      strcpy  (d_rec->loc,  inrec->lc_host);
72              }



73    /*                                      Write  new  database  changes  to  disk  files       *



74              fprintf(ssn_p,  "%s  %s  %d  %d  %d  %d\n",
75                      d_rec->ssn,  d_rec->loc,
76                      d_rec->sunrise, d_rec->sunburn, d_rec->sunspot,
77                      d_rec->hpnotic);

78
79              return  (1);
80    }
```

```
81      /*                          READ_PERS_REC                              *
82      /*                                                                     *
83      /*              This function reads the desired personnel record into the    *
84      /*              structure p_rec.                                       *


85      int read_pers_rec()

86      {
87              if (!p_rec)
88              {
89                      p_rec = (pers_rec *) malloc(sizeof(pers_rec));
90                      p_rec->ssn = (char *) malloc(sizeof(SSN_SIZE));
91                      p_rec->name = (char *) malloc(sizeof(NAME_SIZE));
92                      p_rec->address = (char *) malloc(sizeof(ADDR_SIZE));
93              }

94              if (fscanf(fp2, "%s %s %s",
95                              p_rec->ssn, p_rec->name, p_rec->address) != 3)
96                              return (0);
97              return (1);
98      }




99      /*                          SSN_KEY_1                                  *
100     /*                                                                     *
101     /*              This function is called remotely by an established client.    *
102     /*              The client sends the local host location and personnel filename,* /
103     /*              and this server returns the actual location of the personnel    *
104     /*              file, as well as the accumulated statistical information about   *
105     /*              that perticular file.                                  *


106     dbase_rec *ssn_key_1(input_rec)
107             inputrec                *input_rec;
108     {
109             char *DFILE;

110             DFILE           = (char *) calloc(MAX_REC_LEN+1, sizeof(char));
111             strncpy(DFILE, "filedb.txt",MAX_REC_LEN);


112
113             if (!(fp1 = fopen (DFILE, "r+")))
114                             return ((dbase_rec *) NULL);

115             while (read_dbase(input_rec))
116                             if (!strcmp(d_rec->ssn, input_rec->ssn))
117                                     break;

118             ssn_p = fopen("tempdb2", "w");

119             if feof (fp1)
120             {
121                     fclose (fp1);
122                     return ((dbase_rec *) NULL);
123             }
```

```
124              write_dbase(input_rec);
125              fclose (fp1);
126              fclose(ssn_p);
127              system("/users/andyjank/project1/changefile");
128      return ((dbase_rec *) d_rec);
129    }




130    /*                          GET_REC_1                              *
131    /*                                                                 *
132    /*              This function is called remotely by an established client.   *
133    /*              The client sends the host location and personnel filename, and  *
134    /*              this server returns the personnel file.             *


135    pers_rec *get_rec_1(input_rec)
136              inputrec              *input_rec;

137    {
138              CLIENT    *c2;
139              info_rec info;
140              char *number[20];
141              char *PFILE;
142              pers_rec               *error;

143              PFILE              = (char *) calloc(MAX_REC_LEN+1, sizeof(char));
144              strncpy(PFILE, "D.",MAX_REC_LEN);

145              strcat(PFILE, input_rec->ssn);


146              if (!(fp2 = fopen (PFILE, "r")))
147                      return ((pers_rec *) NULL);

148              while (read_pers_rec())
149                      if (!(strcmp(p_rec->ssn, input_rec->ssn)))
150                              break;

151              if feof (fp2)
152              {
153                      fclose (fp2);
154                      return ((pers_rec *) NULL);
155              }
156              fclose (fp2);



157    /*  This code section contacts the file transfer server (that would  *
158    /*  actually perform the file transfer in a fully implemented        *
159    /*  application.                                                     *

160              info.oldloc  = oldname;
161              info.newloc  = input_rec->h_name;
162              info.filename  = d_rec->ssn;


163              if (!(c2 = clnt_create("hpnotic",FTSPROG, FTSVERS, "tcp")))
```

```
164             {
165               clnt_pcreateerror("hpnotic");
166               exit(1);
167             }

168             trans_1(&info,  c2);

169             return  ((pers_rec  *)  p_rec);
170      }
```

Appendix

E

```
 1    /*                    *** project 1 ***                              * /
 2    /*                                                                   * /
 3    /*                          FTS.X                                    * /
 4    /*                                                                   * /
 5    /*                This file is compiled by RPCgen to produce the     * /
 6    /*                header file for our file transfer server.          * /
 7    /*                                                                   * /


 8    const HOST_SIZE = 255;
 9    const F_NAME_SIZE = 255;

10                        /* Structure that contains the original location of the
11                        /* file, the filename, and the desired new file location* /


12    struct info_rec
13    {
14             string      oldloc<HOST_SIZE>;
15             string      newloc<HOST_SIZE>;
16             string      filename<F_NAME_SIZE>;
17    };




18    program FTSPROG
19    {
20             version FTSVERS
21             {
22                      int TRANS(info_rec) = 1;
23             } = 1;
24    } = 0x20009301;
```

```
 1   /*                      *** project 1 ***                              * /
 2   /*                                                                     * /
 3   /*                      FTS_SVC_PROC.C                                  * /
 4   /*                                                                     * /
 5   /*            This file is compiled by RCPgen to produce the           * /
 6   /*      file transfer server executable.  The executable itself * /
 7   /*      does not actually perform the file transfer to the new    * /
 8   /*      server; it only produces a message informing the user     * /
 9   /*      that this is where the actual transfer would take place.* /
10   /*                                                                     * /


11   #include <stdio.h>
12   #include <string.h>
13   #include <rpc /rpc.h>
14   #include "fts.h"



15   int *trans_1(h_info)

16           info_rec              *h_info;

17   {
18           static int num;
19           num = 1;

20
21           if(strcmp (h_info->oldloc, h_info->newloc) == 0)
22             printf("\nThe file %s will remain \nat the server %s.",
23                     h_info->filename, h_info->oldloc);
24
25           else
26             printf("\nThe file %s has been moved\nfrom %s to %s.",
27                     h_info->filename, h_info->oldloc, h_info->newloc);
28     return (&num);
29   }
```

```
1    #!  /bin /csh  −f


2    #                          SSN_IDENT:                            #
3    #                                                                #
4    #          This  UNIX  script  takes  the  input  SSN  and  creates  a  file  #
5    #          that  includes  all  non−matches  of  that  SSN.              #


6    grep  −v  `cat  ssn.txt`   /users /andyjank /project1 /filedb.txt  >  tempdb
```

```
1    #!   /bin /csh  −f


2    #                         CHANGEFILE:                                    #
3    #                                                                        #
4    #          This  UNIX  script  updates  the  database  file.             #
5    #                                                                        #


6    cat  tempdb2  >>  tempdb
7    cat  tempdb  >!  filedb.txt
```

```
1    #! /bin /csh  −f

2    #                         FIND_HNAME:                              #
3    #                                                                  #
4    #          This UNIX script finds the hostname of the client.      #

5    echo `hostname` >! hname.txt
```